Timing analysis by model checking

Dimitri Naydich and David Guaspari [*]
Odyssey Research Associates
33 Thornwood Drive, Suite 500
Ithaca, NY 14850-1250
naydich@clarityconnect.com
davidg@oracorp.com

Abstract: The safety of modern avionics relies on high integrity software that can be *verified* to meet hard real-time requirements. The limits of verification technology therefore determine acceptable engineering practice. To simplify verification problems, safety-critical systems are commonly implemented under the severe constraints of a cyclic executive, which make design an expensive trial-and-error process highly intolerant of change. Important advances in analysis techniques, such as rate monotonic analysis (RMA), have provided a theoretical and practical basis for easing these onerous restrictions. But RMA and its kindred have two limitations: they apply only to verifying the requirement of schedulability (that tasks meet their deadlines) and they cannot be applied to many common programming paradigms.

We address both these limitations by applying *model checking*, a technique with successful industrial applications in hardware design. Model checking algorithms analyze finite state machines, either by explicit state enumeration or by symbolic manipulation. Since quantitative timing properties involve a potentially unbounded state variable (a clock), our first problem is to construct a finite approximation that is conservative for the properties being analyzed—if the approximation satisfies the properties of interest, so does the infinite model. To reduce the potential for state space explosion we must further optimize this finite model. Experiments with some simple optimizations have yielded a hundred-fold efficiency improvement over published techniques.

1 The safety of hard real-time software

Modern avionics relies fundamentally on high integrity software that meets hard real-time requirements such as schedulability—the guaranty that all tasks meet their deadlines. It is common to implement a high integrity real-time system by means of a cyclic executive, in which programmers explicitly allocate the execution of processes or process fragments to portions of a master control loop. This technique has the strengths of requiring essentially no runtime support and of making schedulability analysis trivial. But the design of a cyclic executive is expensive and time-consuming, relies heavily on trial-and-error rather than systematic design principles, and is highly intolerant of change. Small modifications to individual processes may require complete redesign of the master control loop. In addition, this narrowing of the design space potentially constrains the introduction of automation technologies that could improve both safety and performance.

The alternative to a cyclic executive is some form of preemptive scheduling in which processes are scheduled dynamically. Preemptive scheduling immediately presents two problems: First, static analysis of program behavior becomes much more difficult. Second, the runtime support required to carry out dynamic scheduling must be efficient and must admit an implementation simple enough to satisfy the certification requirements for high integrity systems. Raven [32] is an example of such a runtime.

The best-known analysis technique for preemptive scheduling is Rate Monotonic Analysis (RMA) [19], which applies to a restricted but useful class of systems and reduces schedulability analysis to checking a set of simple algebraic inequalities. However, RMA does not provide information about properties other than schedulability and is not applicable to

[*] This work was partially supported by NASA Langley, contract NAS1-20335

many common programming paradigms: Figure 1 provides an example of such a program. Nor does RMA cover properties other than schedulability.

This paper describes an ongoing investigation of model *checking* as a supplement to RMA. Model checking comprises automated techniques that apply, in principle, to any system representable as a finite state machine. These techniques are of two general kinds: *explicit search* (clever strategies for visiting all possible states) and *symbolic model checking* (combining symbolic execution and automated reasoning). Both styles can be used to analyze properties other than schedulability and systems that do not meet the design restrictions imposed by RMA. Our work shows that model checking can be applied to some systems beyond the reach of current analytical techniques. The technical barrier to making these applications practical and routine is the possibility of state space explosion. We are investigating optimization techniques that generate efficient representations of the system to be analyzed.

1.1 Ravenscar and Raven

The general principles we employ are not tied to any particular implementation, though the details will necessarily depend on the programming language and runtime system being modeled. The Ravenscar Profile [8] defines a set of Ada tasking features rich enough to support (among other things) rate monotonic scheduling, but requiring a minimal runtime. Ravenscar is supported by the Raven runtime, developed at Aonix to meet the highest FAA certification standards for safety critical systems. The tasking subset we consider can be regarded as a generalization of Ravenscar, together with a technical requirement, which we call *frame synchronization*, that reduces nondeterminism by eliminating arbitrary task phasings. Thus, the analysis we propose can be directly applied to real systems.

1. The main features of the Ravenscar subset are as follows:
2. The number of tasks, and the base priority of each, is fixed and statically determined.
3. Scheduling is preemptive, using the priority ceiling protocol.
4. Tasks interact only through protected objects. No more than one task may ever be queued on the entry of any protected call. (This limit on the size of the entry queues is a dynamic requirement that cannot in general be enforced by syntactic restrictions.)
5. Task behavior is deterministic.

Figure 1, based on an example from [16], illustrates a simple Ravenscar program to which RMA does not apply. Three sensors periodically sample flight data and send it via a bounded buffer to an analyzer that periodically reads the data from the buffer. The buffer is implemented as a protected object containing a protected entry for writing data and a protected procedure for reading it. A read from an empty buffer returns some conventional value. The buffer's *write* entry blocks the sensors from writing when the buffer is full. The protected *read* procedure blocks the analyzer from reading while the buffer is being written to. (We make the *read* operation a procedure rather than an entry because Ravenscar forbids protected objects with more than one entry. That is why *read* does not block on any empty buffer, but reads some conventional value.) RMA does not apply because each of the periodic sensor tasks contains a protected entry call, at which it can be blocked.

1.2 Model-checking real-time properties

Many existing models for real-time systems are based on timed automata [2] or, more generally, hybrid automata [1]. These models contain state variables that represent the values of real-time clocks. Notice that a direct model of time, by means of a variable containing the current value of the clock, leads to an infinite state space, since the clock may increase without bound. Some form of temporal abstraction is required. The abstraction used to analyze hybrid automata is to represent *regions*—sets of states—symbolically, via logical formulas. Symbolic manipulation of such formulas [20] is the heart of model checking tools such as [4].

In [10], Corbett presents a two-stage construction that models real-time Ada tasking programs (together with the supporting runtime) as hybrid automata. The first stage translates a program to a transition system representing the possible interleavings of the tasks' execution. The second stage captures the timing constraints of the program by transforming the transition

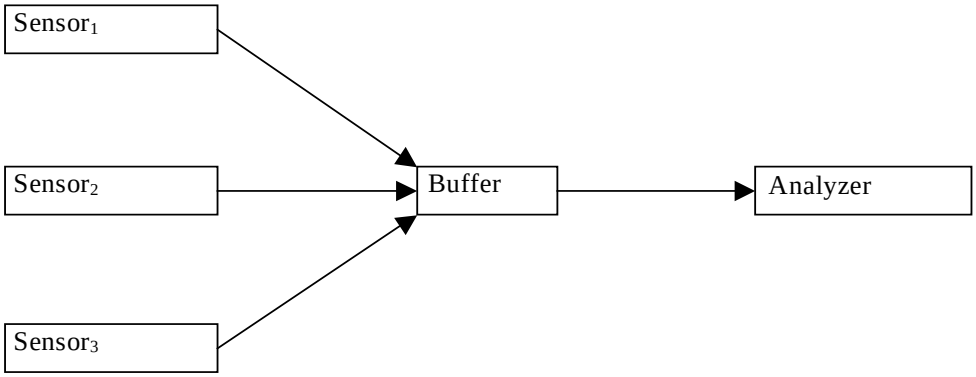

Figure 1: A Ravenscar example to which RMA does not apply

system into a hybrid automaton. This hybrid automaton is then analyzed by the HyTech verifier [11], which can be regarded as symbolic model checker.

In [16], we developed a method for constructing models of real-time tasking programs in Promela [12], a language for specifying communicating sequential processes. The program's tasks and the runtime system are represented as Promela processes. The frame synchronization requirement mentioned above allows us to eliminate the real-time clocks from the system's model altogether and thereby to represent the system as a simple transition system rather than a hybrid automaton. We introduce state variables to keep track of upper and lower bounds on the completion time of each process, and perform a "dynamic abstraction" of these time-related state variables to make the state space finite. In essence, the pair of completion times for each process defines the region of states in which the process is running. This representation is much simpler than representation by a logical formula. We then analyzed the Promela program with the Spin verifier [12].

Many other formal models have been proposed for concurrent real-time systems [3]. These include Petri nets [14], timed automata [2], timed process algebras [17], and real-time logics [13]. For the most part, these models are intended to represent specification, not implementation. In [5], general timed automata are extended to represent such implementation details as the assignment of tasks to processors, priorities, worst-case execution times of operations, and scheduling policies. Our model compares to [5] much as it does to [10].

2 A simple illustration

This section uses a trivial example to show how the "dynamic time abstraction" of [16] can be combined with reduction techniques from [10] and illustrate its effectiveness. Although there are enough differences that a quantitative comparison is not strictly scientific, we obtain a hundred-fold advantage over [10] in both speed and memory usage and a ten-fold advantage over [16].

2.1 A schedulability problem

Consider two periodic, non-interacting tasks, A and B, run on a single processor under preemptive scheduling. Task A has higher priority than B. Although this trivial tasking pattern can be analyzed by RMA, it allows us to illustrate essential features of our proposed strategy and to perform a simple comparison with Corbett's analysis via a hybrid automaton model.

A code skeleton is given in Figure 2. We assume that the variable *StartTime* records the value of the system clock at some moment after the tasks have been initialized but before they start running. In effect, this implements the frame synchronization assumption. *StartTime* can be initialized to satisfy the assumption by using a simple Ada coding idiom given in [16]. The code fragments <*statements1*> and <*statements2*> implement periodic activities whose functionality is irrelevant to the tasks' timing. Let *estimA* and *estimB* be upper bounds on the amount of CPU time necessary to execute the bodies of the loops in task *A* and task *B* respectively. We assume that CPU time is the tasks' only shared resource. The parameters

```
task A is
    pragma Priority(20);
end A;

task body A is
        nextA: Time = StartTime;
begin
        loop
         <statements1>
        nextA := nextA + periodA;
        delay until nextA;
        end loop;
end A;
```

```
task B is
    pragma Priority(10);
end B;

task body B is
        nextB: Time = StartTime;
begin
        loop
         <statements2>
        nextB := nextB + periodB;
        delay until nextB;
        end loop;
end B;
```

Figure 2 : A two-task problem

and *periodA* and *periodB* define the periods of task *A* and task *B*. Execution of "**delay until** *t*" blocks a task until the system clock has value *t*. If task *A* reaches its "**delay until** *nextA*" statement when the clock time is greater than *nextA*, then task A has missed a deadline. We can characterize a missed deadline for task *B* similarly.

With this definition of deadline, we analyze the schedulability of tasks *A* and *B* in terms of the task periods *periodA* and *periodB*, and the CPU time estimates *estimA* and *estimB*. As noted, RMA handles the problem easily, but the point of the example is to exhibit simple optimization strategies that can dramatically improve the efficiency of analysis by model checking.

2.2 A discrete model

In the program of Figure 2, the only variables affecting the timing behavior of the program are *nextA* and *nextB*. They are the only data variables represented in our model.

To model the program's control state, we completely abstract from the code fragments within the task loops. We represent the fragments as abstract actions whose executions take time, and whose executions can be preempted by higher priority actions. We model execution of tasks *A* and *B* as periodic invocations of these abstract actions.

In [16] we represented the runtime and each task as a separate process. As observed in [10], this simple-minded representation introduces unnecessary states because the actions of the runtime are so tightly coupled to the actions of the tasks. That is, we know a strong *invariant* that permits a more efficient abstraction of the state space. Because task *A* has higher priority than task *B*, we can partition the system states as follows: task *A* can be either running or blocked by its "delay until" statement; task *B* can be running, or blocked by its "delay until" statement, or preempted by task *A*; and the system as a whole enters an error state if either task misses a deadline. Thus, we represent the status of the program by introducing a variable *runtime_status* that can have the following symbolic values: *runningA_preemptedB*, *blockedA_runningB*, *blockedA_blockedB*, *runningA_blockedB*, and *missed_deadline*.

We also introduce several variables to model timing information:

1. The integer variables *lb* and *ub* specify lower and upper bounds for the clock time at which the currently executing abstract action will (if not preempted) complete. The values of these time bounds vary dynamically, according to the program's control flow.
2. The integer variable *delta* contains an upper bound for the CPU time needed to complete the currently executing abstract action. When a preempted action resumes its execution the value of *delta* will typically be revised to reflect the progress made before preemption.
3. The integer variable *preemptB*, called the *preemption bound*, stores the value of *delta* when task *B* is preempted by A.

We specify the schedulability requirement by

asserting that the runtime status *missed_deadline* never occurs:

```
Invariant "hard deadline"
    ! runtime_status = missed_deadline
```

The states and transitions of our model are shown graphically in Figure 3. We define the effect of each transition using the notation of the Murphi model-checker [33]. The meaning of

guard ==> Begin <statements> End

is that the transition may take place when the boolean *guard* is true; and, if it does take place, the effect on the state variables is defined by the Pascal-like code in *statements.* If several transitions may take place, then the choice of which transition to fire is non-deterministic. (Even if the Ada code is deterministic our model may be a conservative, non-deterministic, approximation.) The simple model shown here does not represent the overhead attributable to runtime actions such as preempting a task or restoring the state of a preempted task. Those costs are accounted for explicitly in [16].

Figure 4 provides definitions for three representative transitions: 1, 2, and 4. Transition rules 1 and 2 describe the program's behavior when *A* is running and *B* is preempted. Rule 4 describes one of the possible behaviors of the system when task *A* is blocked and task *B* is running—namely, the possibility that task *A* may preempt task *B*.

Rule 1: If the upper estimate of the clock time for completing task *A* is greater than or equal to the next deadline—that is, $ub \geq nextA+periodA$—then it is possible that *A may* miss its deadline; and therefore a deadline violation *will be* reported. Our model is a conservative approximation of the program. The program will satisfy any invariant satisfied by the model, but the converse need not be true.

Rule 2: If $ub < nextA+periodA$, this iteration of task *A* will meet its deadline. Transition 2 represents the successful completion of *A*, after which *A* becomes blocked until the beginning of its next period, and hands off to task *B* (as reflected by changing the value of *runtime_status* to *blockedA_runningB*). To do the necessary bookkeeping, the other state variables are modified as follows:

- *nextA*, the next clock time at which task *A* becomes ready to run, is incremented by the value of its period,
- *delta*, the estimate of the remaining CPU time to complete task *B*, is restored to the preemption bound of B,
- *ub*, which now represents an upper estimate of the clock time at which task *B* will complete, is increased by *delta*,
- since the preemption of B has now been accounted for, we reset *preemptB* to zero.

Rule 4: The guard for transition 4 represents the following possibility: task *B* will, if not preempted, meet its deadline; but task *A* becomes ready before the action of task *B* completes and therefore preempts *B*. Among the actions of rule 4, the interesting new feature is a call to procedure *time_wrap*, which is essential for making our model finite.

The state variables *nextA, nextB, lb,* and *ub* are regularly incremented. If we allowed them to increase without bound our model's state space would be infinite. However, the presence or absence of a deadline violation depends only on the *relative* values of these variables, not on their absolute values. Therefore, the relevant timing behavior of our model does not change if we recalibrate by simultaneously decreasing *nextA, nextB, lb,* and *ub* by the same amount. Procedure *time_wrap* does the recalibration, decrementing all these variables by the current value of *lb*. Our transition rules will invoke *time_wrap* immediately after any increment to *lb*. This is a form of rolling, dynamic time abstraction.

This recalibration strategy will succeed in bounding the values of these variables if the *differences* between the values of *nextA, nextB, lb,* and *ub* are bounded. It is shown in [16] that, for all the executions of the model in which no deadline is missed, the absolute values $|nextA-lb|$,$|nextB-lb|$, and $|ub-lb|$ will all be less than $2*max(periodA, periodB)$. Therefore we can statically restrict the range of the time variables to $-MAX$ .. MAX, where $MAX=2*max(periodA, periodB)$. To be more precise, if there is a deadline violation in the infinite model (from which all occurrences of *time_wrap* have been deleted), then there is a deadline violation in the recalibrated model, and it will be detected before

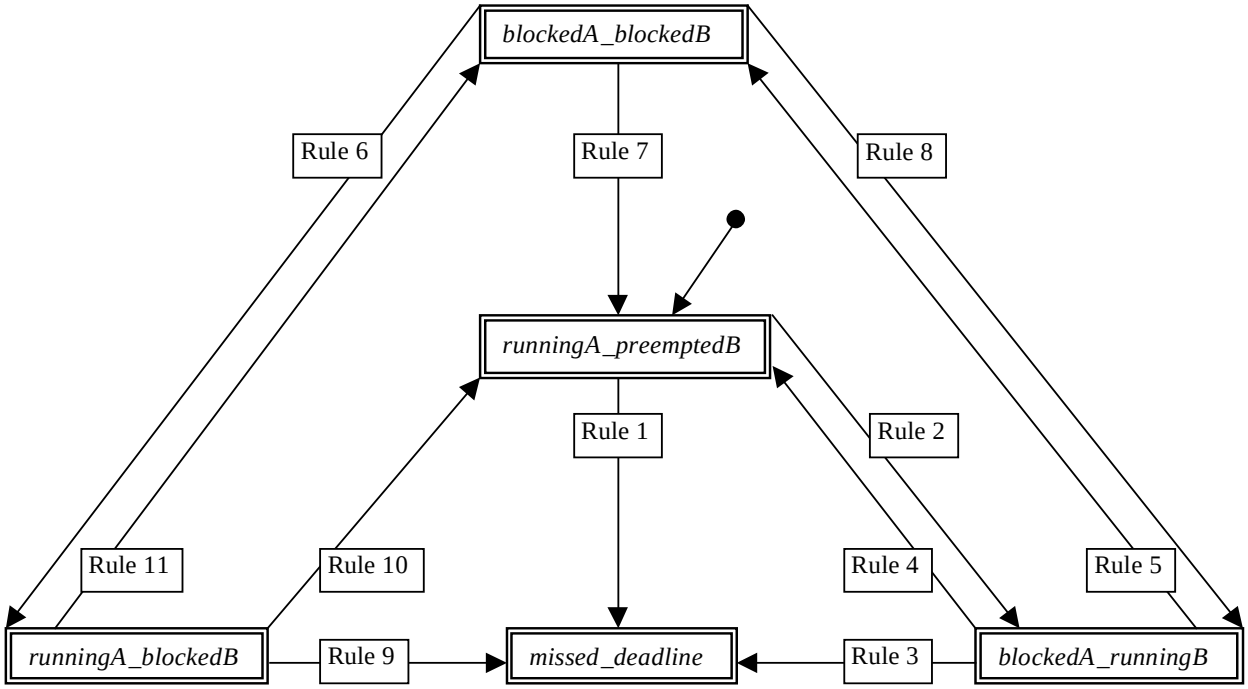

Figure 3: The transition system model

```
    Rule "1"
    runtime_status=
runningA_preemptedB
    & ub >= nextA + periodA
        ==>
    Begin

    runtime_status          :=
missed_deadline;
    End;
```

```
    Rule "2"
    runtime_status          =
runningA_preemptedB
    & ub < nextA + periodA
            ==>
    Begin
    nextA := nextA + periodA;
    runtime_status          :=
blockedA_runningB;
    delta := preemptB;
    ub := ub + preemptB;
    preemptB := 0;
    End;
```

```
    Rule "4"
    runtime_status = blockedA_runningB
    & ub < nextB + periodB
    & nextA < ub
    ==>
    Begin

    runtime_status:=runningA_preempte
dB;
    preemptB := (ub - nextA < delta) ?
                        (ub - nextA) :
delta;
    delta := estimA;
    lb := nextA;
    ub := nextA + estimA;
    time_wrap();
    End;
```

Figure 4: Representative transition rules

execution of the model attempts an update that puts these variables out of range.

2.3 A comparison

Our experiment analyzed the example of section 2.1 in three ways: We applied Murphi to the transition system defined in section 2.2; we applied HyTech to the hybrid automaton constructed by the methods of [10] alone; we applied SPIN to the model constructed by the methods of [16] alone. The comparison with [10], for various values of the parameters, is shown in the charts below.

We suspect that that the advantage of these

estimA=5, periodA = 10, estimB = 10, periodB = 30	*Transition system*	*Hybrid automaton*
Number of states/regions	11	8
CPU time (sec)	0.10	0.24
Memory used	1K	0.82M

estimA = 29, periodA = 59, estimB = 61, periodB = 181	*Transition system*	*Hybrid automaton*
Number of states/regions	1002	480
CPU time (sec)	0.10	13.73
Memory used	25K	4.53M

estimA = 167, periodA = 353, estimB = 313, periodB = 997	*Transition system*	*Hybrid automaton*
Number of states/regions	5013	2700
CPU time (sec)	0.40	106.95
Memory used	163K	20.13M

Figure 5 : A comparison

optimizations will increase as the timing constraints become more complex, because manipulating integers is more efficient than manipulating linear formulas with integer coefficients. We cannot quantify how much of the difference might be attributable to the fact that Murphi is a more mature tool than HyTech.

The advantage over [16] is not quite so dramatic—the improvement is one order of magnitude, not two.

2.4 Other properties

This section briefly considers problems other than schedulability. The model and the size of the state space depend on the property analyzed. For example, in the terminology of Figure 2, it is easier to analyze the assertion that "Both tasks always meet their deadlines" than to analyze the assertion "Task *B* always meets its deadlines," because uncertainty about the behavior of *A* would add nondeterminism to the model. Since the tasks of Figure 2 do not interact (except implicitly, via preemption) there is not much to ask about this example aside from its schedulability.

When tasks do interact, things become more interesting. The Ravenscar rules require that no more than one task be waiting on the entry of any protected call. The main purpose of this requirement is to avoid the overhead of maintaining queues. In general, it is undecidable whether a program meets the requirement, though compliance could be guaranteed by making severe static semantic restrictions on the code. The Raven runtime raises an error dynamically if execution ever violates the requirement. Thus, it is important to be able to check this rule by static analysis. A schedulability model of the kind suggested in this section already encodes enough information in its state to answer this question. Analysis of the length of entry queues is insensitive to the recalibration trick.

Deadlock freedom is another interesting

question that should be amenable to our techniques. The priority ceiling protocol itself suffices to guarantee that a certain class of tasking programs cannot deadlock, but the general question is undecidable. (This problem is also insensitive to recalibration.)

2.5 Limitations

We might hope for a divide-and-conquer approach whereby knowing that the system is schedulable—for example, in cases where RMA is applicable—might permit us to produce a simpler model with which we might verify other properties. However, if the precise timing behavior of the program is necessary to guarantee those properties, we must represent that behavior in our model and therefore encode the schedulability problem within it. In effect, verifying schedulability is automatically part of verifying any property at all. Unfortunately, the intricacies and timing of task interleavings are the principal source of state space explosion.

Our experience thus far suggests that the effectiveness of our methods will depend more on the underlying set of tasking primitives than on a discipline restricting the patterns in which they are used. Interrupts are especially interesting, and present special problems. In the model of [16] we found that code with interrupts typically resulted in a state space explosion. Symbolic model checking may be applicable to this case. On the other hand, several tasking constructs omitted by the Ravenscar Profile seem amenable to model checking analysis: absolute delay statement; rendezvous; select statements.

3 More realistic examples

This section briefly describes the application of our model-checking techniques to more realistic examples. We summarize experiments using the methods of [16] on a modest work station, which we have not had the opportunity to repeat with the optimizations proposed above. These examples employ the main Ravenscar tasking constructs such as "**delay until**" statements, protected procedures and entries, interrupts, and sporadic tasks triggered by interrupts.

The modeling of interrupts and sporadic tasks is the most complicated part of the model of [16]. Conceptually, a sporadic task is triggered by an interrupt and must complete its response interrupt within a specified *response time*. Each interrupt is characterized by its *minimum interarrival time*—the minimum time between two consecutive occurrences of the interrupt. The minimum interarrival time and the response time for each interrupt are parameters of the model.

To implement sporadic tasks we use an Ada idiom required by the Ravenscar programming discipline: The response to an interrupt *I* is performed by a sporadic task *T* whose body is a loop. The head of that loop is a call on a protected entry *E*, so that task *T* is blocked at the head of the loop so long as entry barrier of *E* is false; and the last act of the loop is to reset the entry barrier of *E* to true. The text of an Ada program binds interrupt *I* to a protected procedure *P*, which will be executed by the runtime whenever *I* occurs; and, in this programming idiom, *P* must be implemented so that its only effect is to change the entry barrier of *E* from false to true. Thus, when interrupt *I* occurs, the runtime executes *P*, which sets the barrier of *E* to true; that unblocks task *T*, which performs the response to the interrupt, resets the barrier of *E* to false, and becomes suspended.

We permit tasks to contain both "**delay until**" statements and entry calls. For our purposes, a task containing a "**delay until**" statement is *periodic*. A *sporadic* task contains a call on a protected entry whose barrier is set by an interrupt handler. Since we impose no upper limit on the interrupt interarrival time, a sporadic task cannot be guaranteed to satisfy any periodic deadline. For this reason, sporadic tasks may not contain 'delay until' statements. The Promela code checks that all periodic tasks meet their deadlines and that the response to every interrupt completes within the response time.

We have analyzed several systems containing both periodic and sporadic tasks, all on a SparcServer20 with 64 megabytes of memory.

One is a toy pump control system [29] often used as a benchmark example, which our techniques handled in seconds. With some more complicated systems, however, the model of [16] encountered a state space explosion. We describe two such examples:

1. the Olympus attitude and orbital control system (AOCS) [30],
2. a brewery control program [31].

A pump controller

The pump control system has the following components:

1. four periodic tasks getting data from the four sensors and controlling the pump,
2. a sporadic task, triggered by the interrupt from a high/low water level detector, that controls the pump, and
3. two protected objects for the pump and the interrupt interface.

Verification of this program took 20 seconds.

The AOCS

The AOCS design contains 17 protected objects, 4 sporadic tasks driven by interrupts (with short interarrival times), and 10 periodic tasks (with relatively long periods). We were able to verify a reduced version with all 10 periodic tasks and only one sporadic task (roughly 1.5 hours of computation). Adding a second sporadic task resulted in a state space explosion that SPIN could not handle.

A Brewery controller

Our techniques successfully identified a timing error in the brewery control program, but the analysis required some abstractions performed by hand, not merely the "standard" abstractions used to represent the pump controller.

The brewery control program contains no interrupts. It consists of an alarm task suspended on a protected entry, several short-period tasks, and one long-period task that calculates a "pattern temperature." One of the short-period tasks compares the actual temperature to the pattern and, if the difference between the temperatures is too great, opens the entry barrier to trigger the alarm. We model the decision about whether to trigger the alarm as a completely nondeterministic event (a conservative approximation).

We may eliminate the long-period task altogether if we assume that the pattern temperature is constant. Under that assumption (also conservative) our methods took 6 minutes of computation to find a timing violation.

If we do not assume that the pattern temperature is constant, the combination of a long-period task with a short-period task nondeterministically triggering another task results in a state space explosion (as explained below).

The size of our model's state space is proportional to S^P, where:

1. P is the number of possible patterns of the periodic tasks' arrival times. (A task *arrives* whenever it begins a new period.). P is roughly proportional to (M/D), where M is the least common multiple of the task periods and interrupt interarrival times, and D is their greatest common divisor.
2. S is the average number of non-deterministic choices exercised by the model during the execution of any one pattern of arrival times. A common source of non-determinism is the runtime process controlling task preemption. However, this nondeterminism is usually restricted, since the control-delegating conditions in the runtime process are often mutually exclusive. Thus, the runtime process does not contribute much to the size of S. On the other hand, nondeterministic behavior in a short-period task will increase S, since this behavior is exercised in the many patterns where the task is running.

Our problem with the brewery control program is that the short-period task nondeterministically triggers the alarm, which increases S. We can still analyze the program if P is low, but including the long-period task increases P. This combination increases S^P sufficiently to cause a state explosion.

As for the interrupts, in [16] we model each interrupt by a Promela process representing a "quasi-task" that makes calls on the protected procedure that is its handler. The behavior of such a task is in many respects similar to the behavior of a periodic task that non-deterministically executes the interrupt handler and has a period equal to the interrupt's minimum interarrival time.

4 Future research

Our primary technical problem is how to optimize the model for efficient model-checking. The optimizations described in section 2—the runtime status abstractions, the encoding of regions as pairs of integers—are specific to our problem domain and to the kinds of properties being analyzed. There is an extensive literature on general-purpose algorithms for abstractions and optimizations of untimed transition systems, and on the automated discovery of invariants. (See, for example, [21-24]). Future research will consider the applicability of that literature to our problem.

Symbolic model checking is another possibility for dealing with state space explosion. Problems that do not yield to explicit search

techniques can sometimes be solved by symbolic model checking (and vice versa). The state-machine model accepted by a symbolic model checker is typically quite low-level and constrained. Not all symbolic model checkers permit variables of integer type. But some, such as WSMV [9], are able to treat integers and certain integer operations symbolically by using special encoding techniques that permit efficient representation of addition and integer comparisons, and those are precisely the arithmetical operations our methods require. Thus, WSMV is a promising engine for extending our results with symbolic model checking.

References

[1] R. Alur, C. Coucoubetis, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science* 138, pp. 3-34, 1995.

[2] R. Alur and D. L. Dill. A theory of timed automata**.** *Theoretical Computer Science* 126, pp. 183-235, 1994.

[3] R. Alur and T.A. Henzinger. Logics and Models of Real Time: A Survey. In *Real-Time: Theory in Practice, REX Workshop*, LNCS 600, pp. 74-106, 1991.

[4] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic Symbolic Verification of Embedded Systems. *IEEE Transactions on Software Engineering* 22, pp. 181-201, 1996.

[5] K. Brink, J. Katwijk, R. Spelberg, and H. Toetenel. Analyzing schedulability of Astral specifications using extended timed automata. *Proceedings of the Third International Euro-Par Conference*, LNCS 1300, pp. 1290-1297, Springer-Verlag, 1997.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond, *Information and Computation*, vol. 98, pp. 142-170, 1992.

[7] A. Burns. Preemptive Priority-Based Scheduling: An Appropriate Engineering Approach. In *Advances in Real-Time Systems*, S. H. Son, Ed.: Prentice Hall, pp. 225-248, 1994.

[8] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar tasking profile for high integrity real-time programs. Proceedings of *Reliable Software Technologies - Ada-Europe'98*, LNCS 1411, pp. 263-275, 1998.

[9] E.M. Clarke, M. Khaira, and X. Zhao. Word-level symbolic model checking: a new approach for verifying arithmetic circuits. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference*, IEEE Computer Society Press, 1996.

[10] J. C. Corbett. Timing analysis of Ada tasking programs. *IEEE Transactions on software engineering*, 22(7), pp. 461-483, 1996.

[11] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer* 1, pp. 110-122, 1997.

[12] G. J. Holzmann. *Design and validation of computer protocols*: Prentice Hall, 1991. (The current version of Spin can be found at http://netlib.bell-labs.com/netlib/spin/whatispin.html.)

[13] C. Ghezzi, D. Mandriolli, and A. Morzenti. Trio: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2), pp. 107-123, 1990.

[14] C. Ghezzi, D. Mandriolli, S. Morasca, and M. Pezze. A unified high-level Petri net model for time-critical systems. *IEEE Transactions on software engineering*, 17(2), 1991.

[15] K. G. Larsen, P. Pettersson and Wang Yi. UPPAAL in a Nutshell. In *Springer International Journal of Software Tools for Technology Transfer* 1(1+2), 1997.

[16] D. Naydich and D. Guaspari. Analyzing Ravenscar Profile tasks by model checking. Technical report TM-98-0034, Odyssey Research Associates, 1998.

[17] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249-261, June 1988.

[18] G. Romanski *Safety critical software handbook*. Aonix, 1997.

[19] L. Sha, R. Rajkumar, and S. S. Sathaue. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. *Proceedings of IEEE*, vol. 82, pp. 68--82, 1994.

[20] S. Wolfram. *Mathematica: A system for doing mathematics by computer*. Adisson-Wesley, 1988.

[21] The SAL Group. The SAL Intermediate Language.

[22] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. To appear in *Formal Methods of System Design*.

[23] S. Bensalem,Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically.
[24] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), 1994.
[25] M. Bickford and D. Naydich. Hardware verification technology transfer: Application of formal methods and modeling to the ARM6. Tech. Rep. TM98-0021, ORA, 1998.
[26] Sast User Manual (version 0.2). Odyssey Research Associations, 1997.
[27] Z. Chen and D. Hoover. TableWise, a decision table tool. *Proceedings of the Tenth Annual Conference on Computer Assurance (Compass '95)*.
[28] D. Guaspari, C. Marceau, and W. Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, vol. 16, no. 9, September, 1990. Reprinted in proceedings of the First International Workshop on Larch, Springer-Verlag, 1993.
[29] A. Burns and A. J. Wellings, *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems*: Elsevier, 1995.
[30] A. Burns, A. J. Wellings, C. M. Bailey, and E. Fyfe, "The Olympus Attitude and Orbital Control System: A Case Study in Hard Real-Time System Design and Implementation," Proceedings of *Ada sans frontiers -- 12th Ada-Europe Conference*, LNCS 688, pp. 19-35, 1993.
[31] G. Romanski, "Ada, Concurrency and a Safety Critical Subset," Personal communications, 1998.
[32] "Raven Fact Sheet", Aonix, 1999.
[33] David L. Dill, Andreas J. Drexler, Alan J. Hu and C. Han Yang, "Protocol Verification as a Hardware Design Aid," 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society, pp. 522-525.